HW1

Emanuele Iaccarino 2192710
roberto magno mazzotta 2200470
Augusto de Luzenberger Milnernsheim, 1601657
Jacopo caldana 2212909

2025-02-11

Part 1: When Brutti and Spinelli collide

1. Background: Classification 101

The problem of predicting a discrete random variable Y from a random vector X is called classification.

$$(Y_1, X_1), (Y_2, X_2), ..., (Y_n, X_n) \quad \text{with} \quad Y_i \in \{0, 1\}$$

(binary) response variable and

$$\mathbf{X}_i \in \mathcal{X} \subset \mathbb{R}^d$$

a vector of explanatory variables (or features, or covariates)

A classification rule or classifier is then any function

$$\eta : \mathcal{X} \mapsto \{0, 1\}.$$

When we observe a new cov vector, X say, we then predict the response Y to be (X).

Just as an example, consider the fake n = 12 data-points displayed in the figure below. The covariate X = (X1, X2) is 2–dimensional and the outcome

$$\mathbf{Y} \in \{0, 1\} = \{\triangle, \bigcirc\}.$$

Also shown is a linear classification rule represented by the solid line. For suitable values of the parameters (, 1, 2) to be estimated from the data, this rule is of the form

$$\eta(\mathbf{x}) = \begin{cases} 1, & \text{if } \alpha + \beta_1 x_1 + \beta_2 x_2 > 0, \\ 0, & \text{otherwise.} \end{cases}$$

Everything above the line is classified as a 0 and everything below the line is classified as a 1. These two groups are perfectly separated by the linear decision boundary (see the definition below); you probably won't see such a simple pattern in real datasets.

Ok, now that we know what a classifier is, how do we evaluate its performance? Let's start by introducing the following (quite natural) loss function, its associated risk, and finally the empirical counterpart of the latter:

The 0/1 loss of a classifier

$$\eta(\cdot) : L(Y, \eta(\mathbf{X})) = \mathbb{I}(Y \neq \eta(\mathbf{X})).$$

The risk or true error rate of a classifier

$$\eta(\cdot) \text{ is: } R(\eta) = \mathbb{E}(L) = \mathbb{P}\left(\{Y \neq \eta(\mathbf{X})\}\right).$$

The empirical error rate or training error rate is:

$$\widehat{R}_n(\eta) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}\left(\eta(\mathbf{X}_i) \neq Y_i\right).$$

To keep going, let's look a bit closer at the structure of the regression function (i.e. the conditional expected value of Y given X) once specialized to this context

$$r(\mathbf{x}) = \mathbb{E}(Y \mid \mathbf{X}) \underset{Y \text{ binary}}{=} P(Y = 1 \mid \mathbf{X} = \mathbf{x}) \underset{\text{Bayes'TH}}{=} \frac{f(\mathbf{x} \mid Y = 1)P(Y = 1)}{f(\mathbf{x} \mid Y = 1)P(Y = 1) + f(\mathbf{x} \mid Y = 0)P(Y = 0)}$$

$$= \frac{\pi_1 \cdot f_1(\mathbf{x})}{\pi_1 \cdot f_1(\mathbf{x}) + (1 - \pi_1) \cdot f_0(\mathbf{x})}$$

$$\text{where} \quad f_0(\mathbf{x}) = f(\mathbf{x} \mid Y = 0), \quad f_1(\mathbf{x}) = f(\mathbf{x} \mid Y = 1), \quad \pi_1 = P(Y = 1).$$

Nice. Last round of definitions and then the result: The set

$$\mathcal{X}(\eta) = \{\mathbf{x} \in \mathcal{X} : \mathbb{P}(Y = 1 \mid \mathbf{X} = \mathbf{x}) = \mathbb{P}(Y = 0 \mid \mathbf{X} = \mathbf{x})\}$$

is called the decision boundary.

We define the Bayes classification rule or Bayes Classifier $\eta^*(\cdot)$ as:

$$\eta^*(\mathbf{x}) = \begin{cases} 1, & \text{if } r(\mathbf{x}) \geq \frac{1}{2} \\ 0, & \text{otherwise} \end{cases}$$

which is equivalent to:

$$\eta^*(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbb{P}(Y = 1 \mid \mathbf{X} = \mathbf{x}) > \mathbb{P}(Y = 0 \mid \mathbf{X} = \mathbf{x}) \\ 0, & \text{otherwise} \end{cases}$$

Furthermore, using Bayes' Theorem, this can be rewritten as:

$$\eta^*(\mathbf{x}) = \begin{cases} 1, & \text{if } \pi_1 f_1(\mathbf{x}) > (1 - \pi_1) f_0(\mathbf{x}) \\ 0, & \text{otherwise} \end{cases}$$

And here we go with the main result we mentioned also in class:

The Bayes Classifier is optimal, that is, if (·) is any other classifier, then R(*) R(). The Bayes Classifier is "just" a theoretical benchmark, that clearly depends on unknown quantities that we need to estimate/learn from data to make it practical. But please notice: the Bayes Classifier has nothing whatsoever to do with Bayesian Inference. We could estimate/learn the Bayes Classifier using either Bayesian or frequentist methods.

Your job - Part 1

Suppose that (Y, X) are random variables with

$$Y \in \{0, 1\} \quad \text{and} \quad X \in \mathbb{R}.$$

Suppose that

$$(X \mid Y = 0) \sim \text{Unif}(-3, 1) \quad \text{and} \quad (X \mid Y = 1) \sim \text{Unif}(-1, 3).$$

Further suppose that

$$\mathbb{P}(Y = 0) = \mathbb{P}(Y = 1) = \frac{1}{2}$$

Find (with pen and paper) the Bayes classification rule *(x).

We want to find the **Bayes classification rule** for

$$\eta^*(x).$$

From the cheat-sheet, we know:

$$f_X(x) = \frac{1}{b-a}\mathbb{I}(x) \quad \text{for } x \in [a,b] \quad \text{where} \quad a, b \in \mathbb{R}.$$

$$f_X(x) = \frac{1}{4}\mathbb{I}_{[-3,1]}(x) \quad \text{for } X, Y \in \{0,1\}$$

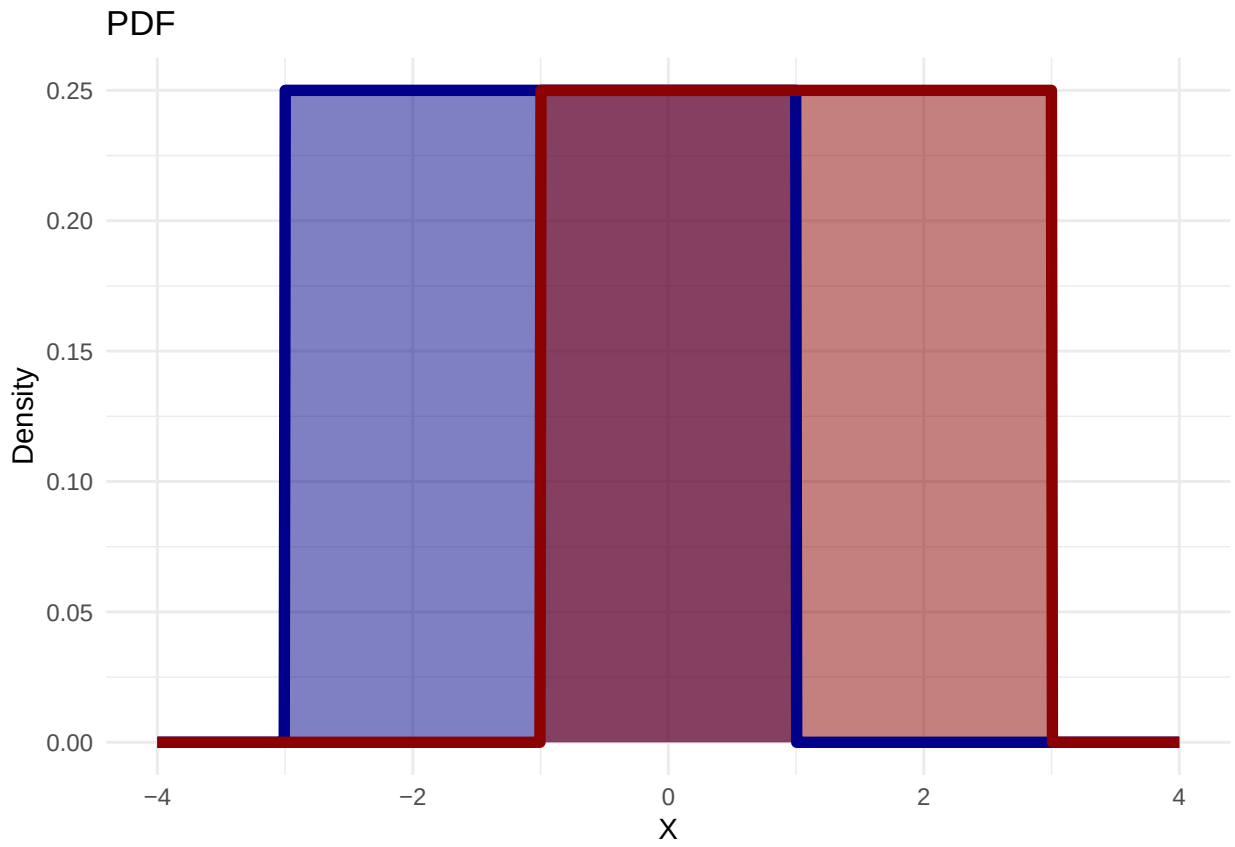
From the given information:

$$p(x \mid Y = 0) = \begin{cases} \frac{1}{4}, & \text{for } x \in [-3,1] \\ 0, & \text{otherwise} \end{cases}$$

Similarly,

$$p(x \mid Y = 1) = \begin{cases} \frac{1}{4}, & \text{for } x \in [-1,3] \\ 0, & \text{otherwise} \end{cases}$$

```r
library(ggplot2)
library(tidyr)
library(dplyr)

# PDF
x_vals <- seq(-4, 4, length.out = 1000)  # Range of x values for plotting

df <- tibble(
  x = x_vals,
  f0 = ifelse(x_vals >= -3 & x_vals <= 1, 1/4, 0),   # p(x | Y=0)
  f1 = ifelse(x_vals >= -1 & x_vals <= 3, 1/4, 0)    # p(x | Y=1)
) %>%
  pivot_longer(cols = c(f0, f1), names_to = "label", values_to = "density")

# Plot
ggplot() +
  geom_ribbon(data = df, aes(x = x, ymin = 0, ymax = density, fill = label), alpha = 0.5) +
  geom_line(data = df, aes(x = x, y = density, color = label), size = 2) +  # Thick lines
  labs(title = "PDF", x = "X", y = "Density") +
  scale_fill_manual(values = c("f0" = "darkblue", "f1" = "darkred"), guide = "none") +
  scale_color_manual(values = c("f0" = "darkblue", "f1" = "darkred"), guide = "none") +
  theme_minimal()
```



I removed the legends from the plot since not really clear, explanation is below:

- darkblue + darkred for p(x | Y = 0)
- darkorange + darkred for p(x | Y = 1)
- darkred focus on the intersection of the two areas p(x | Y = 0) p(x | Y = 0)

$$r(x) = \frac{p(x \mid Y=1)P(Y=1)}{p(x \mid Y=1)P(Y=1) + p(x \mid Y=0)P(Y=0)}$$

Since we know that:

$$P(Y=0) = P(Y=1) = \frac{1}{2}$$

we substitute:

$$r(x) = \frac{\frac{1}{2}p(x \mid Y=1)}{\frac{1}{2}p(x \mid Y=1) + \frac{1}{2}p(x \mid Y=0)}$$

which simplifies to:

$$r(x) = \frac{p(x \mid Y=1)}{p(x \mid Y=1) + p(x \mid Y=0)}$$

Case I: If $x < -3$ or $x > 3$, then no valid $p(x)$ exists.

Case II: If $x \in [-3, -1]$, then:

$$p(x \mid Y=0) = \frac{1}{4}, \quad p(x \mid Y=1) = 0$$

$$r(x) = \frac{0}{\frac{1}{4} + 0} = 0 \Rightarrow P(Y=1 \mid X=x) = 0$$

Case III: If $x \in [-1, 1]$, then:

$$p(x \mid Y = 0) = \frac{1}{4}, \quad p(x \mid Y = 1) = \frac{1}{4}$$

$$r(x) = \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{4}} = \frac{0.5}{1} = 0.5 \Rightarrow P(Y = 1 \mid X = x) = 0.5$$

Case IV: If $x \in (1, 3]$, then:

$$p(x \mid Y = 0) = 0, \quad p(x \mid Y = 1) = \frac{1}{4}$$

$$r(x) = \frac{\frac{1}{4}}{\frac{1}{4} + 0} = 1 \Rightarrow P(Y = 1 \mid X = x) = 1.$$

Summary:

$$r(x) = \begin{cases} 0, & x \in [-3, -1) \\ 0.5, & x \in [-1, 1] \\ 1, & x \in (1, 3] \\ 0, & \text{otherwise} \end{cases}$$

$$\eta^*(x) = \begin{cases} 1, & \text{if } r(x) \geq 0.5 \\ 0, & \text{otherwise} \end{cases}$$

$$r(x) = 0 \Rightarrow x < -1 \Rightarrow \eta^*(x) = 0$$
$$r(x) = 0.5 \Rightarrow x \in [-1, 1] \Rightarrow \eta^*(x) = 0$$
$$r(x) = 1 \Rightarrow x \in (1, 3] \Rightarrow x > 1 \Rightarrow \eta^*(x) = 1$$

$$\text{so:} \quad \eta^*(x) = \begin{cases} 0, & x \leq 1 \\ 1, & x > 1 \end{cases} \Rightarrow \text{Decision boundary: } x = 1$$

2. Simulate n = 1000 data from the joint data model p(y, x) = p(x | y) * p(y) described above, and then:

- Plot the data (or a subset for clarity) together with the regression function r(x) that defines *(x)

```
library(ggplot2)
```

```
Warning: il pacchetto 'ggplot2' è stato creato con R versione 4.3.3
```

```
library(tidyr)
```

```
Warning: il pacchetto 'tidyr' è stato creato con R versione 4.3.3
```

```
library(dplyr)
```

```
Warning: il pacchetto 'dplyr' è stato creato con R versione 4.3.3


Caricamento pacchetto: 'dplyr'
I seguenti oggetti sono mascherati da 'package:stats':

    filter, lag
I seguenti oggetti sono mascherati da 'package:base':

    intersect, setdiff, setequal, union
```

```r
# Define constants
set.seed(22) # my lucky number
n =1000  # Number of data points for train
m = 10000 # Number of data points for test
```

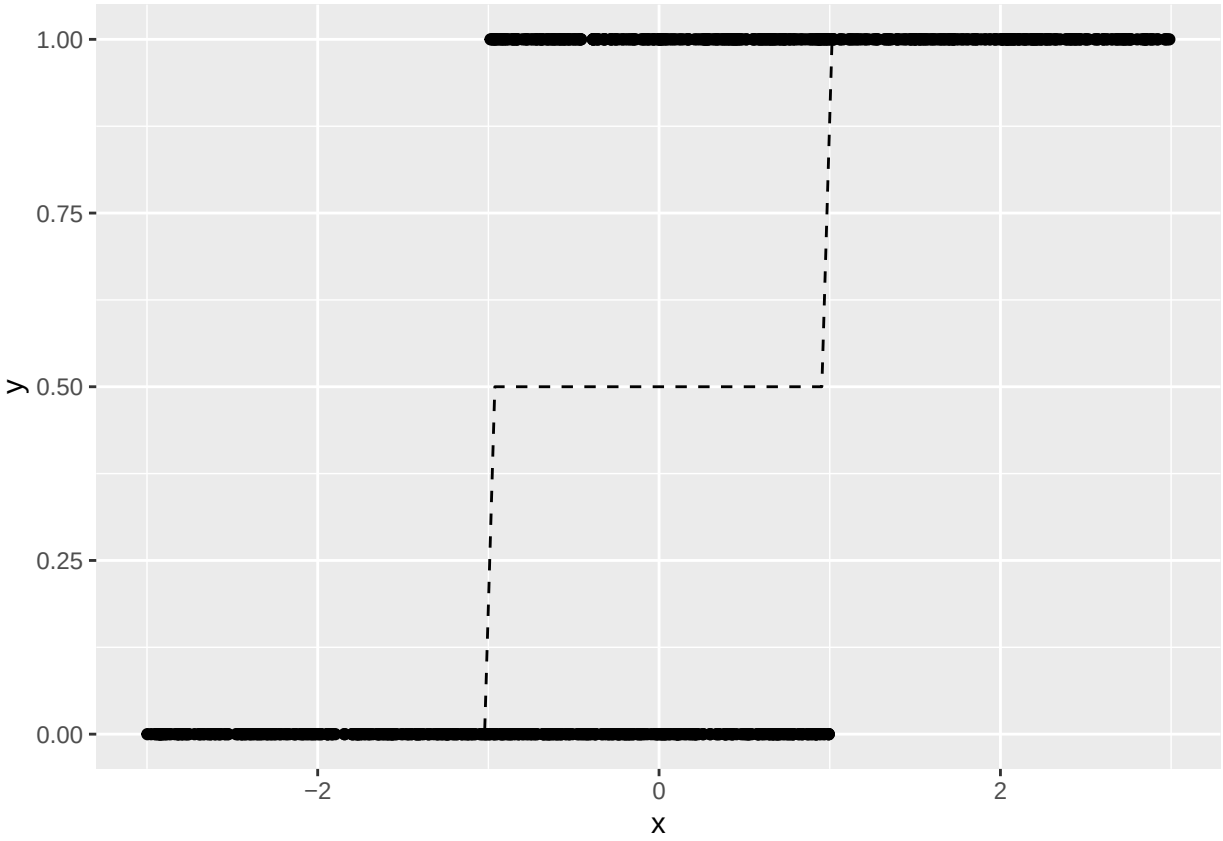
```r
# Distributions

# Density function for X | Y = 0 (Uniform(-3, 1))
f0 = function(x) ifelse(x >= -3 & x <= 1, 1 / 4, 0)
# Density function for X | Y = 1 (Uniform(-1, 3))
f1 = function(x) ifelse(x >= -1 & x <= 3, 1 / 4, 0)
```

```r
# Simulate Data
generate_data = function(sample_size) {
y = rbinom(sample_size, 1, 0.5)  # Simulate labels Y with P(Y = 1) = P(Y = 0) = 0.5
x = ifelse(y == 0, runif(sample_size, -3, 1), runif(sample_size, -1, 3))  # Simulate X base
data = data.frame(x = x, y = y)
}
```

```r
# Regression function r(x)
r = function(x) {
  num = 0.5 * f1(x)  # Numerator: P(Y = 1) * f1(x)
  denom = 0.5 * f1(x) + 0.5 * f0(x)  # Denominator: Weighted sum of densities
  num / denom  # Conditional probability P(Y = 1 | X = x)
}
```

```r
x_seq = seq(-3, 3, length.out = 1000)  # Sequence of x values for plotting
r_vals = r(x_seq)  # Compute r(x) values for the sequence
```

```r
# Scatter plot with Bayes Classifier decision boundary
ggplot(data, aes(x = x, y = y)) +
  geom_point() +  # Plot the data points
  stat_function(fun = r, linetype = "dashed") +  # Add r(x) curve
  labs(
    title = "Bayes Classifier Decision Boundary",
    x = "X (Feature)",
    y = "Y (Class)"
  ) +
  theme_minimal()
```



This plot shows the data points we geneerated and r(x) function used as decision rule for Bayes classifier

- Evaluate the performance of the Bayes Classifiers *(x) on this simple (only 1 feature!) data

```
# Bayes classification rule
bayes_classifier = function(x) {
  ifelse(r(x) > 0.5, 1, 0)  # Classify as 1 if r(x) > 0.5, else 0
}
```

```
m = 10000 # Number of data points per Test set
data_test = generate_data(m) # test data
#summary(r(data_test$x))

data_test$bayes_pred = bayes_classifier(data_test$x)
# Bayes Classifier
confusion_matrix = table(Predicted = data_test$bayes_pred, Actual = data_test$y)
accuracy_bayes = sum(diag(confusion_matrix)) / nrow(data_test)
print(confusion_matrix)
```

```
         Actual
Predicted    0    1
        0 5000 2506
        1    0 2494
```

```
print(accuracy_bayes)
```

```
[1] 0.7494
```

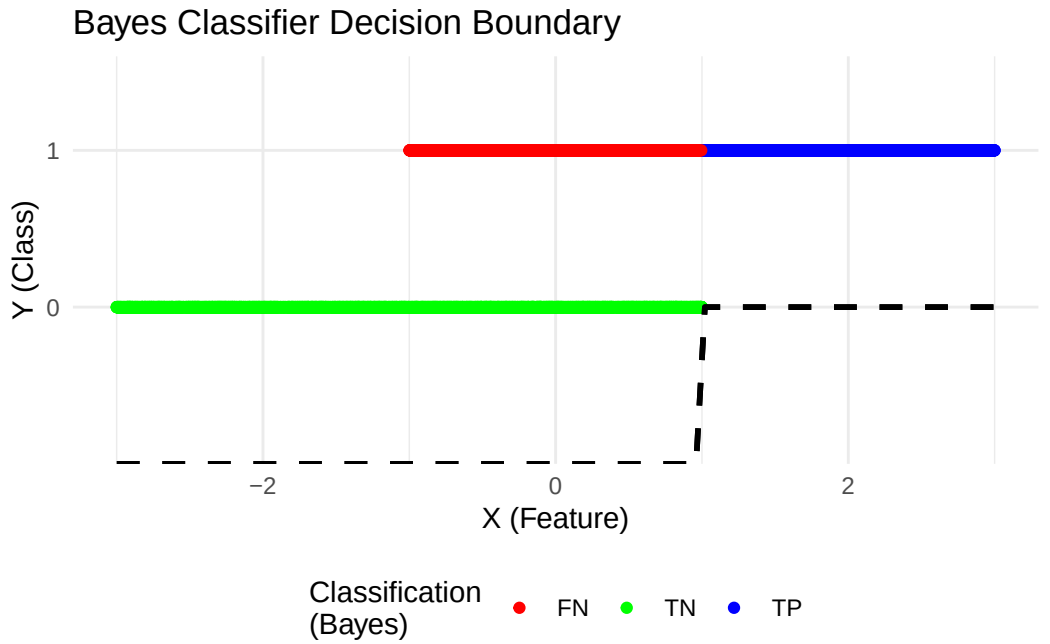
The Bayes classifier perfectly predicts all positive cases (Y=1) as Y=1, which is why there are no false negatives (FN=0).

However, it misclassifies a portion of Y=0 cases as Y=1, leading to some false positives (FP=2529)

```
# Classification results
data_test$classification_bayes = with(data_test, ifelse(
  bayes_pred == 1 & y == 1, "TP",
  ifelse(bayes_pred == 0 & y == 0, "TN",
         ifelse(bayes_pred == 1 & y == 0, "FP", "FN"))))

ggplot(data_test, aes(x = x, y = as.factor(y))) +
  geom_point(aes(color = classification_bayes)) +
  stat_function(fun = bayes_classifier, color = "black", linetype = "dashed", size = 1) +
  labs(
    title = "Bayes Classifier Decision Boundary",
    x = "X (Feature)",
    y = "Y (Class)",
    color = "Classification\n(Bayes)"
  ) +
  scale_color_manual(values = c("TP" = "blue", "TN" = "green", "FP" = "orange", "FN" = "red
  theme_minimal() +
  theme(legend.position = "bottom")
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```

```
Warning: Multiple drawing groups in `geom_function()`
i Did you use the correct group, colour, or fill aesthetics?
```



Regarding the following discussion lead us to some doubts: https://elearning.uniroma1.it/mod/forum/discuss.php?d=246287

First of all the claims done were wrong, we prove it empirically and theorically:

```
set.seed(2)
# Range of thresholds
threshold_values <- seq(0.25, 0.75, by = 0.05)
```

```r
# Store results
results <- data.frame(Threshold = numeric(), Accuracy = numeric())

# Iterate through each threshold and compute accuracy
for (thresh in threshold_values) {
  data_test$bayes_pred <- ifelse(r(data_test$x) > thresh, 1, 0)
  accuracy <- sum(data_test$bayes_pred == data_test$y) / nrow(data_test)
  results <- rbind(results, data.frame(Threshold = thresh, Accuracy = accuracy))
}

print(results)
```

```
   Threshold Accuracy
1       0.25   0.7459
2       0.30   0.7459
3       0.35   0.7459
4       0.40   0.7459
5       0.45   0.7459
6       0.50   0.7494
7       0.55   0.7494
8       0.60   0.7494
9       0.65   0.7494
10      0.70   0.7494
11      0.75   0.7494
```

We observe two distinct levels of accuracy: - for thresholds below 0.5 - for thresholds 0.5 and above The value of one section is either higher or lower than the other according by the seed choice.

The sudden change from one section to the other occurs precisely at 0.5, indicating that the decision boundary at x=1, derived analytically, is indeed the optimal threshold in terms of accuracy.

Why this happen?? In case II we have computed this:

$$\text{If} \quad x \in [-1, 1], \text{ then:} \quad p(x \mid Y = 0) = \frac{1}{4}, \quad p(x \mid Y = 1) = \frac{1}{4}$$

$$r(x) = \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{4}} = \frac{0.5}{1} = 0.5 \Rightarrow P(Y = 1 \mid X = x) = 0.5$$

So this is the region where changing the threshold redistributes errors between false positives (FP) and false negatives (FN), but does not improve accuracy.

The threshold does not change the total number of mistakes but only moves errors from one category (FP) to another (FN). To be more clear the section with thresholds 0.5 and above minimize FP to 0, while the other section minimize FN to 0, setting our decision boundary to x=-1

- Apply any other practical classifier of your choice to these data and comparatively comment its performance (with respect to those of the Bayes classifiers). Of course those n training data should be used for training and validation too (in case there are tuning-parameters).

Initially we used a CV approach: We have a limited amount of data and we needed un unbiased estimate of performance before final testing to reduce overfitting to a single train-test split

After carefully reading the following discussion:

https://elearning.uniroma1.it/mod/forum/discuss.php?d=247114

The possibility to add an additional fixed evaluation set, where m is the number of data points for test set that we set to 10000 to realistically measures generalization. The choice of m depends on the trade-off between variance and

computational efficiency. A larger test set m reduces the variance of this estimate, giving a more stable and accurate performance measure, that empirically will converge to the true expected accuracy of the classifier.

Since we are working with a simple one-dimensional feature space

$$X \in \mathbb{R}$$

n=1000 is already sufficient for training many simple models (e.g., logistic regression, decision trees).

```
# Training Data
data_train = generate_data(n)

# Logistic regression for comparison
logit_model = glm(y ~ x, data = data_train, family = binomial)
data_test$logit_prob = predict(logit_model, newdata = data_test, type = "response")
data_test$logit_pred = ifelse(data_test$logit_prob > 0.5, 1, 0)

confusion_matrix_logit = table(Predicted = data_test$logit_pred, Actual = data_test$y)
accuracy_logit = sum(diag(confusion_matrix_logit)) / nrow(data_test)
print(confusion_matrix_logit)
```

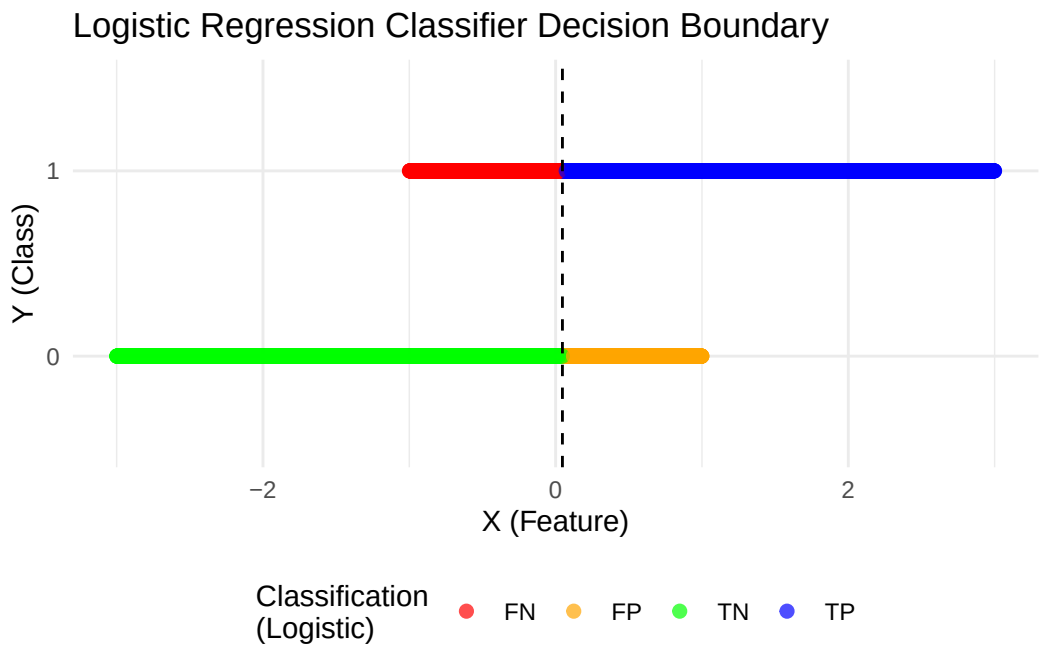
```
         Actual
Predicted    0    1
        0 3773 1341
        1 1227 3659
```

```
print(accuracy_logit)
```

```
[1] 0.7432
```

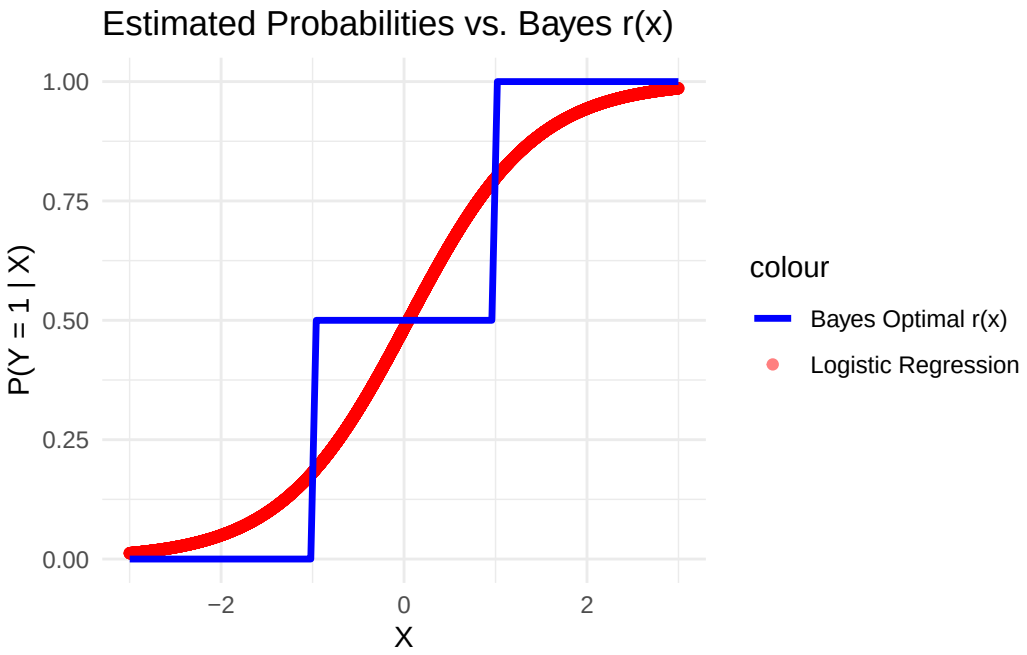
Logistic regression makes a trade-off between false positives and false negatives, performing better in identifying Y=0 correctly compared to Bayes.

```
logit_decision_boundary <- -coef(logit_model)[1] / coef(logit_model)[2]

# Classification results
data_test$classification_logit <- with(data_test, ifelse(
  logit_pred == 1 & y == 1, "TP",
  ifelse(logit_pred == 0 & y == 0, "TN",
         ifelse(logit_pred == 1 & y == 0, "FP", "FN"))))

ggplot(data_test, aes(x = x, y = as.factor(y), color = classification_logit)) +
  geom_point(size = 2, alpha = 0.7) +
  geom_vline(xintercept = logit_decision_boundary, color = "black", linetype = "dashed") +
  labs(
    title = "Logistic Regression Classifier Decision Boundary",
    x = "X (Feature)",
    y = "Y (Class)",
    color = "Classification\n(Logistic)"
  ) +
  scale_color_manual(values = c("TP" = "blue", "TN" = "green", "FP" = "orange", "FN" = "red
  theme_minimal() +
  theme(legend.position = "bottom")
```



By a theorical point of view, Logistic Regression, since being a data-driven model, perform better with limited data or noisy data, it smooth out the decision boundary providing a better trade off between Precision and Recall. The Bayes Classifier instead depends on knowing the exact distributions of X|Y, and in reality this happens really rarely! It maximize the true positive rate (TPR) at the cost of false positives (FP)

```
ggplot(data_test, aes(x = x)) +
  geom_point(aes(y = logit_prob, color = "Logistic Regression"), alpha = 0.5) +
  stat_function(fun = r, aes(color = "Bayes Optimal r(x)"), size = 1.2) +
  labs(title = "Estimated Probabilities vs. Bayes r(x)",
       x = "X", y = "P(Y = 1 | X)") +
  scale_color_manual(values = c("Logistic Regression" = "red", "Bayes Optimal r(x)" = "blue
  theme_minimal()
```



- The Bayes classifier is deterministic given the problem setup(there are sharp transitions at x=1 and x=1, where the probability suddenly jumps).
- Logistic Regression assumes a smooth probability transition because it models P(Y=1X) using a sigmoid function, making it more robust in real-world scenarios where exact distributions are unknown.

Question 3 Since you are simulating the data, you can actually see what happens in repeated sampling. Hence, repeat the sampling M = 10000 times keeping n = 1000 fixed (a simple for-loop will do it), and redo the comparison. Who's the best now? Comment.

```
# Repeat sampling M times
M = 10000 # Repetitions
bayes_accuracies = numeric(M)
logit_accuracies <- numeric(M)
m = 10000

for (i in 1:M) {
  # Generate TRAINING data
  y_train = rbinom(n, 1, 0.5)
  x_train = ifelse(y_train == 0, runif(n, -3, 1), runif(n, -1, 3))
  data_train = data.frame(x = x_train, y = y_train)

  # Generate TEST data
  y_test = rbinom(m, 1, 0.5)
  x_test = ifelse(y_test == 0, runif(m, -3, 1), runif(m, -1, 3))
  data_test = data.frame(x = x_test, y = y_test)

  # Bayes Classifier
  predicted_bayes = bayes_classifier(data_test$x)
  bayes_accuracies[i] = mean(predicted_bayes == data_test$y)

  # Logistic Regression
  logit_model = glm(y ~ x, data = data_train, family = binomial)
```

```
  data_test$logit_prob = predict(logit_model, newdata = data_test, type = "response")
  data_test$logit_pred = ifelse(data_test$logit_prob > 0.5, 1, 0)
  logit_accuracies[i] = mean(data_test$logit_pred == data_test$y)

# Edit: incredible how R is slow to perform for loops
}
mean(bayes_accuracies)
```

```
[1] 0.7500002
```

```
mean(logit_accuracies)
```

```
[1] 0.750003
```

```
sd(bayes_accuracies)
```

```
[1] 0.004356389
```

```
sd(logit_accuracies)
```

```
[1] 0.004290117
```

Over repeated simulations, on average the performance are similar, probably with a greater value of n(# Data points) and/or of M(# Repetition for simulation) they should converge to the same accuracy score (logistic regression will increasingly approximate the true distributions, approaching Bayes' optimal performance).

Running Montecarlo repetitions reduce the variability caused by the seed choice, in fact this could have been misleading (in our first trial Bayes slightly overperform Logistic in term of accuracy, so we could have possibly wrongly claimed that one is better than the other)

```
#    min(P(Y=0) * f0(x), P(Y=1) * f1(x)) dx
# Since P(Y=0) = P(Y=1) = 0.5, we compute:
#    min(0.5 * f0(x), 0.5 * f1(x)) dx
integrand =  function(x) pmin(0.5 * f0(x), 0.5 * f1(x))

# Compute the Bayes error using numerical integration
bayes_error = integrate(integrand, lower = -1, upper = 1)$value
# We integrate only over the region where the two densities overlap (-1,1).

theoretical_bayes_accuracy =  1 - bayes_error

bayes_error
```

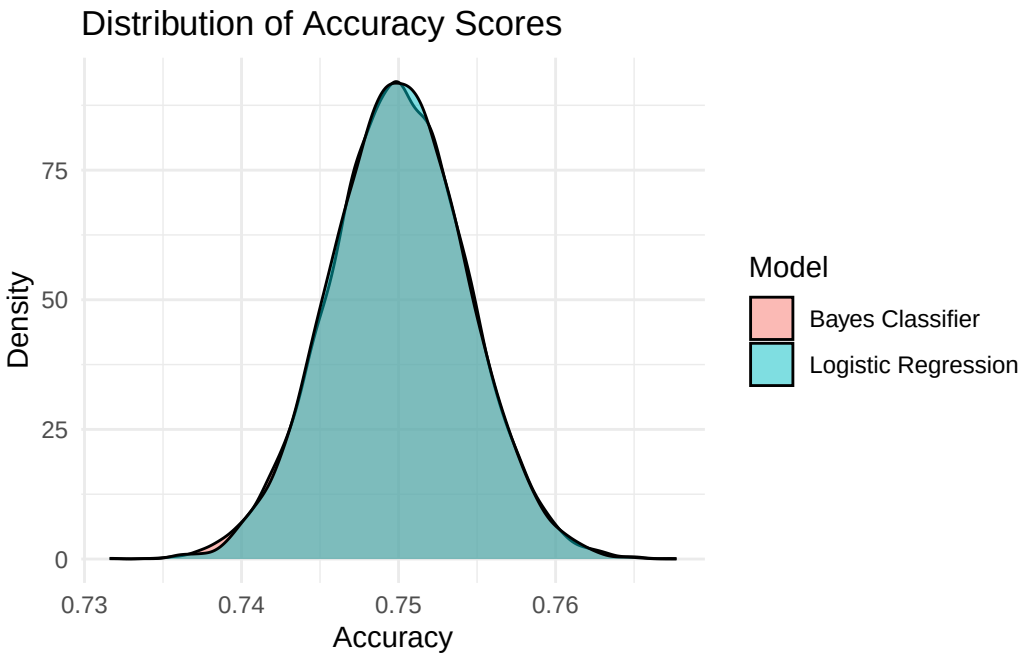
```
[1] 0.25
```

```
theoretical_bayes_accuracy
```

```
[1] 0.75
```

Empirical results match the theorical one

```
accuracy_df = data.frame(
  Accuracy = c(bayes_accuracies, logit_accuracies),
  Model = rep(c("Bayes Classifier", "Logistic Regression"), each = M)
)

ggplot(accuracy_df, aes(x = Accuracy, fill = Model)) +
  geom_density(alpha = 0.5) +
  labs(title = "Distribution of Accuracy Scores", x = "Accuracy", y = "Density") +
  theme_minimal()
```



The distribution is centered around the mean and the value of standard deviation are quite low; using a test set derived by the same distribution but with a larger number of data points allowed us to have pretty consistent results

I am really fussy so we perform a paired t-test to determine if there is a statistically significant difference in their accuracy.

```
# Paired T-Test
t_test_result <- t.test(bayes_accuracies, logit_accuracies, paired = TRUE)
print(t_test_result)
```

```
    Paired t-test

data:  bayes_accuracies and logit_accuracies
t = -0.055636, df = 9999, p-value = 0.9556
alternative hypothesis: true mean difference is not equal to 0
95 percent confidence interval:
 -1.003646e-04  9.482458e-05
sample estimates:
mean difference
      -2.77e-06
```

P value suggest us that we fail to reject the null hypothesis, meaning there is no significant difference in accuracy between the two classifiers. t-value and confidence interval confirming that any observed difference in accuracy is negligible.

Just out of curiosity, at first glance both the scores seams to follow a normal distribution as evidenced by the bell-shaped curve and symmetry, but is that really true?

We check with the Shapiro-Wilk normality test

```
ks.test(bayes_accuracies, "pnorm", mean = mean(bayes_accuracies), sd = sd(bayes_accuracies)
```

```
Warning in ks.test.default(bayes_accuracies, "pnorm", mean =
mean(bayes_accuracies), : i legami non dovrebbero essere presenti per il test
di Kolmogorov-Smirnov
```

```
    Asymptotic one-sample Kolmogorov-Smirnov test

data:  bayes_accuracies
D = 0.0086583, p-value = 0.4416
alternative hypothesis: two-sided
```

```
ks.test(logit_accuracies, "pnorm", mean = mean(logit_accuracies), sd = sd(logit_accuracies)
```

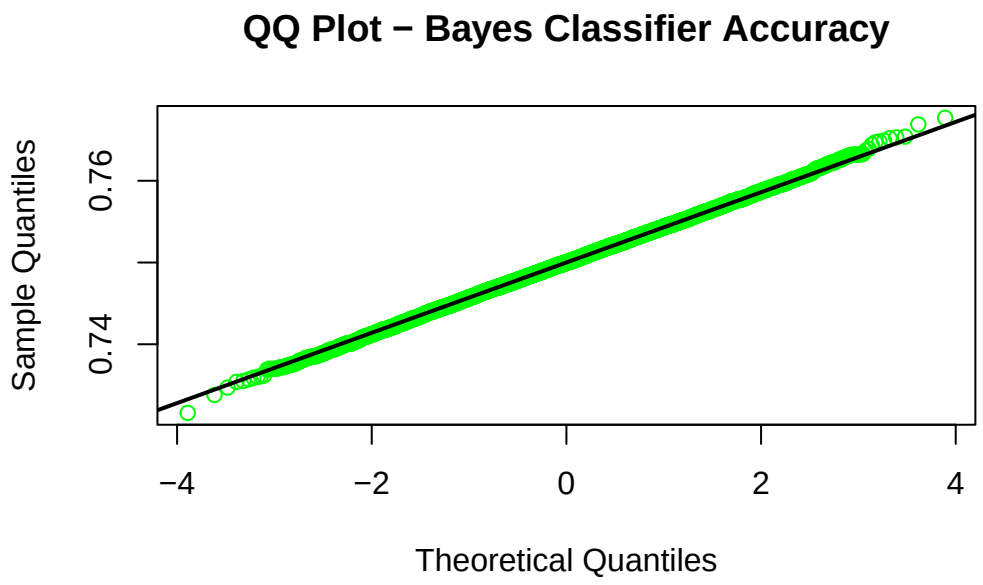
```
Warning in ks.test.default(logit_accuracies, "pnorm", mean =
mean(logit_accuracies), : i legami non dovrebbero essere presenti per il test
di Kolmogorov-Smirnov
```

```
    Asymptotic one-sample Kolmogorov-Smirnov test

data:  logit_accuracies
D = 0.0069185, p-value = 0.7248
alternative hypothesis: two-sided
```

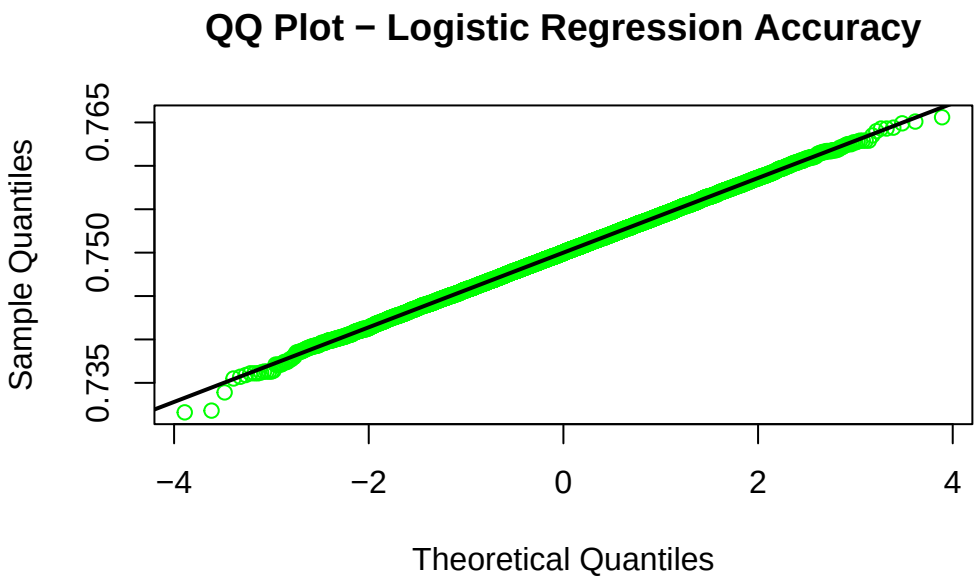
Both the tests not reject the null hypothesis that the data follows a normal distribution.

This test is highly sensitive to small deviations from normality, especially for large sample sizes, despite that, since we computed accuracy as an average over multiple runsit is expected that the empirical distribution of accuracy will approximate a normal distribution due to the CLT (Central Limit Theorem).

```
qqnorm(bayes_accuracies, main = "QQ Plot - Bayes Classifier Accuracy", col = "green")
qqline(bayes_accuracies,lwd = 2)
```




```
qqnorm(logit_accuracies, main = "QQ Plot - Logistic Regression Accuracy", col = "green")
qqline(logit_accuracies, lwd = 2)
```



QQPlot confirm the test results, even though the data points at both tails doesn't follow precisely the diagonal line. Anyway those empirical results confirm approximate normality (almost normal behavior)

The Central Limit Theorem ensures convergence to normality for accuracy scores, given the large number of Monte Carlo repetitions.

But why we did all this? Well first of all we just proved empirically what we studied in theory: - Since errors are tied to discrete cutoffs of uniform distributions, the empirical accuracy distribution can exhibit small skewness or deviations from normality, anyway Monte Carlo estimates of accuracy become approximately normal regardless of those small deviations. - Logistic regression assumes a sigmoid probability function, which means the decision boundary is smoother, in fact since being data driven in this case with a infinite number of data points the distribution converge to a normal distribution

The plots makes everything more clear

In our case, we have "relatively easily distributed data" so we can assume normality but in real scenario this doesn't usually happen and it's important to take this into consideration: - Standard confidence intervals (Mean ± SE) may slightly underestimate uncertainty. - Bootstrap confidence intervals or percentile-based confidence intervals might be more reliable.

```
mean_bayes = mean(bayes_accuracies)
mean_logit = mean(logit_accuracies)
std_err_bayes = sd(bayes_accuracies) / sqrt(length(bayes_accuracies))
std_err_logit = sd(logit_accuracies) / sqrt(length(logit_accuracies))

# Compute Confidence Interval (95% Normal Approximation)
ci_bayes_lower = mean_bayes - 1.96 * std_err_bayes
ci_bayes_upper = mean_bayes + 1.96 * std_err_bayes

ci_logit_lower = mean_logit - 1.96 * std_err_logit
ci_logit_upper = mean_logit + 1.96 * std_err_logit

cat("Bayes Normal CI:", round(ci_bayes_lower, 5), round(mean_bayes, 5), round(ci_bayes_upp
```

```
Bayes Normal CI: 0.74991 0.75 0.75009
```

```r
cat("Logistic Regression Normal CI:", round(ci_logit_lower, 5), round(mean_logit, 5), round
```

```
Logistic Regression Normal CI: 0.74992 0.75 0.75009
```

```r
B = 10000  # Number of bootstrap samples

# Resampling and computing bootstrap means
boot_bayes = replicate(B, mean(sample(bayes_accuracies, replace = TRUE)))
boot_logit = replicate(B, mean(sample(logit_accuracies, replace = TRUE)))

# Compute 2.5% and 97.5% quantiles (Bootstrap CI)
ci_bayes_boot_lower = quantile(boot_bayes, 0.025)
ci_bayes_boot_upper = quantile(boot_bayes, 0.975)

ci_logit_boot_lower = quantile(boot_logit, 0.025)
ci_logit_boot_upper = quantile(boot_logit, 0.975)

cat("Bayes Bootstrap CI:", round(ci_bayes_boot_lower, 5), round(mean_bayes, 5), round(ci_ba
```

```
Bayes Bootstrap CI: 0.74991 0.75 0.75008
```

```r
cat("Logistic Regression Bootstrap CI:", round(ci_logit_boot_lower, 5), round(mean_logit, 5
```

```
Logistic Regression Bootstrap CI: 0.74992 0.75 0.75009
```

There is no difference in our results from the two differents method but it's important to provide significative explanation to every step done to include the reader into the workflow and the thinking process that led us to our final solution

As a challenge we try to use an additional Classifier, we choosed Decision tree for its semplicity, interpretability and the additional challenge of dealing with tunable parameters

```r
library(rpart)
library(rpart.plot)  # For visualizing the tree (
```

```
Warning: il pacchetto 'rpart.plot' è stato creato con R versione 4.3.3
```

```r
# For train and test we use the same train and test set we made previously for the logistic

tree_model = rpart(y ~ x, data = data_train, method = "class")

data_test$tree_pred = predict(tree_model, newdata = data_test, type = "class")

tree_accuracy = mean(data_test$tree_pred == data_test$y)
tree_accuracy
```

```
[1] 0.753
```

Decision Trees are prone to overfitting, so tuning is crucial.

```r
?rpart.control
```

```
avvio in corso del server httpd per la guida ... fatto
```

Key parameters in rpart:

- cp (Complexity Parameter: Controls tree growth. Higher values = smaller trees (prevents overfitting).
- minsplit: Minimum number of observations in a node before splitting.
- minbucket: Minimum number of observations in a terminal node (leaf).
- maxdepth: Maximum depth of the tree. Limits complexity.
- method: Splitting criterion (gini or entropy-based)

To maximize the performance of our Decision Tree based Classifier we need to find the best possible combination of hyperparameters, we simulated a Grid Search approach (available on Python but I didn't find it in R). Anyway we set a grid of values for each hyperparameter we mentioned before (according to prior knowledge and assumption regarding the simplicity of the dataset), we are gonna train and test the model

```
# Number is how many different values we can have for a specific hyperparameter
cp = 10
min_split = 5
max_depth = 6
method = 2
cp * min_split * max_depth * method
```

```
[1] 600
```

and return the combination of hyperparameters that produced the highest accuracy score

Edit: the solution applied for MonteCarlo (using 100% of n as train and test on a new set of data generated by a similar distribution) solved an other problem: In this case we would have needed CV to not base the tuning process on the uncertanty of the split derived by a certain seed, moreover the training time would have been higher. CV is typically used when we need to estimate out-of-sample performance from limited data, which is not the case anymore here.

```
# Define hyperparameter grid
tune_grid <- expand.grid(
  cp = seq(0.001, 0.05, by = 0.005),   # Complexity parameter (pruning)
  minsplit = c(5, 10, 20, 30, 50),     # Minimum samples per split
  maxdepth = c(2, 3, 4, 5, 6, 7),      # Maximum tree depth
  method = c("gini", "information")    # Splitting criterion
)

results <- expand.grid(cp = numeric(), minsplit = numeric(), maxdepth = numeric(), method =

# Grid search
for (i in 1:nrow(tune_grid)) {

  # Train Decision Tree with given parameters
  tree_model <- rpart(y ~ x, data = data_train, method = "class",
                      parms = list(split = tune_grid$method[i]),
                      control = rpart.control(
                                      cp = tune_grid$cp[i],
                                      minsplit = tune_grid$minsplit[i],
                                      maxdepth = tune_grid$maxdepth[i]))

  predictions <- predict(tree_model, newdata = data_test, type = "class")
  accuracy <- mean(predictions == data_test$y)

  results <- rbind(results, data.frame(
    cp = tune_grid$cp[i],
    minsplit = tune_grid$minsplit[i],
    maxdepth = tune_grid$maxdepth[i],
    method = tune_grid$method[i],
    Accuracy = accuracy
  ))
}

best_model <- results %>% arrange(desc(Accuracy)) %>% slice(1)
print(best_model)
```

```
     cp minsplit maxdepth method Accuracy
1 0.006       30        7   gini   0.7557
```

Now we can Train Final Model with Best Parameters

```
best_cp = best_model$cp
best_minsplit = best_model$minsplit
best_maxdepth = best_model$maxdepth

final_tree_model = rpart(y ~ x, data = data_train, method = "class",
                         control = rpart.control(cp = best_cp,
                         minsplit = best_minsplit,
                         maxdepth = best_maxdepth))

data_test$final_tree_pred = predict(final_tree_model, newdata = data_test, type = "class")

final_tree_accuracy = mean(data_test$final_tree_pred == data_test$y)
final_tree_accuracy
```

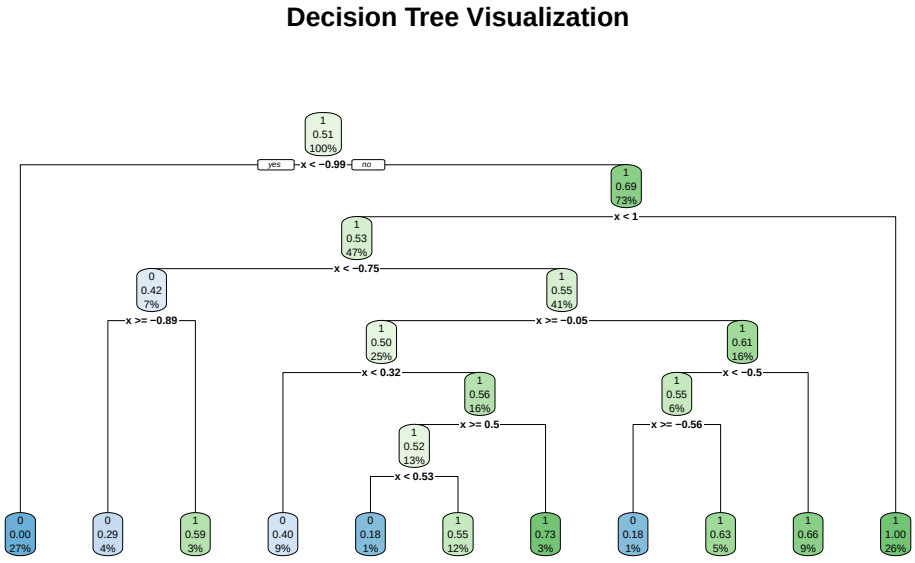
```
[1] 0.7557
```

```
conf_matrix_simple = table(Predicted = data_test$final_tree_pred, Actual = data_test$y)

print(conf_matrix_simple)
```

```
         Actual
Predicted    0    1
        0 3257  732
        1 1711 4300
```

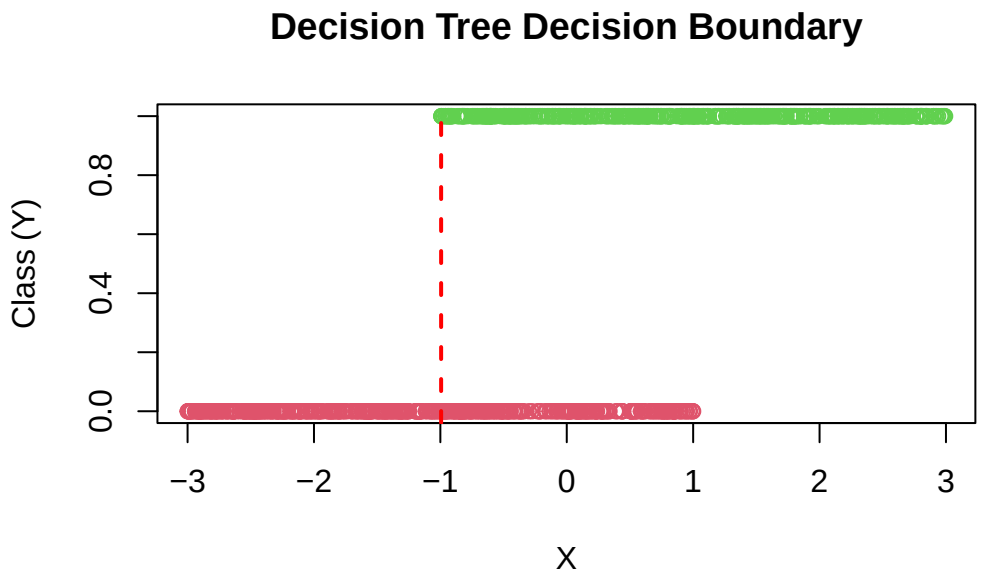
CV allowed us to maximize the

```
rpart.plot(final_tree_model, main = "Decision Tree Visualization")
```




```
x_seq = seq(-3, 3, length.out = 1000)
preds = predict(tree_model, newdata = data.frame(x = x_seq), type = "class")
```

```r
plot(data_train$x, data_train$y, col = as.numeric(data_train$y)+2,
     main = "Decision Tree Decision Boundary", xlab = "X", ylab = "Class (Y)")
abline(v = x_seq[which(diff(as.numeric(preds)) != 0)], col = "red", lwd = 2, lty = 2)
```



The decision boundaries correspond to the splitting thresholds found in the tree visualization(In this case we have more than one decision boundary that indicate the split points where the Decision Tree partitions the feature space)

```r
tree_accuracies = numeric(M)
# Monte Carlo Loop
for (i in 1:M) {
  data_train = generate_data(n)
  data_test = generate_data(m)

  tree_model = rpart(y ~ x, data = data_train, method = "class",
                     control = rpart.control(
                                             cp = best_cp,
                                             minsplit = best_minsplit,
                                             maxdepth = best_maxdepth))

    tree_pred = predict(tree_model, newdata = data_test, type = "class")
  tree_accuracies[i] = mean(tree_pred == data_test$y)
}
```

```r
mean_tree_acc = mean(tree_accuracies)
se_tree_acc = sd(tree_accuracies) / sqrt(M)

ci_tree_lower = mean_tree_acc - 1.96 * se_tree_acc
ci_tree_upper = mean_tree_acc + 1.96 * se_tree_acc

cat("Decision Tree Mean Accuracy:", round(mean_tree_acc, 5), "\n")
```

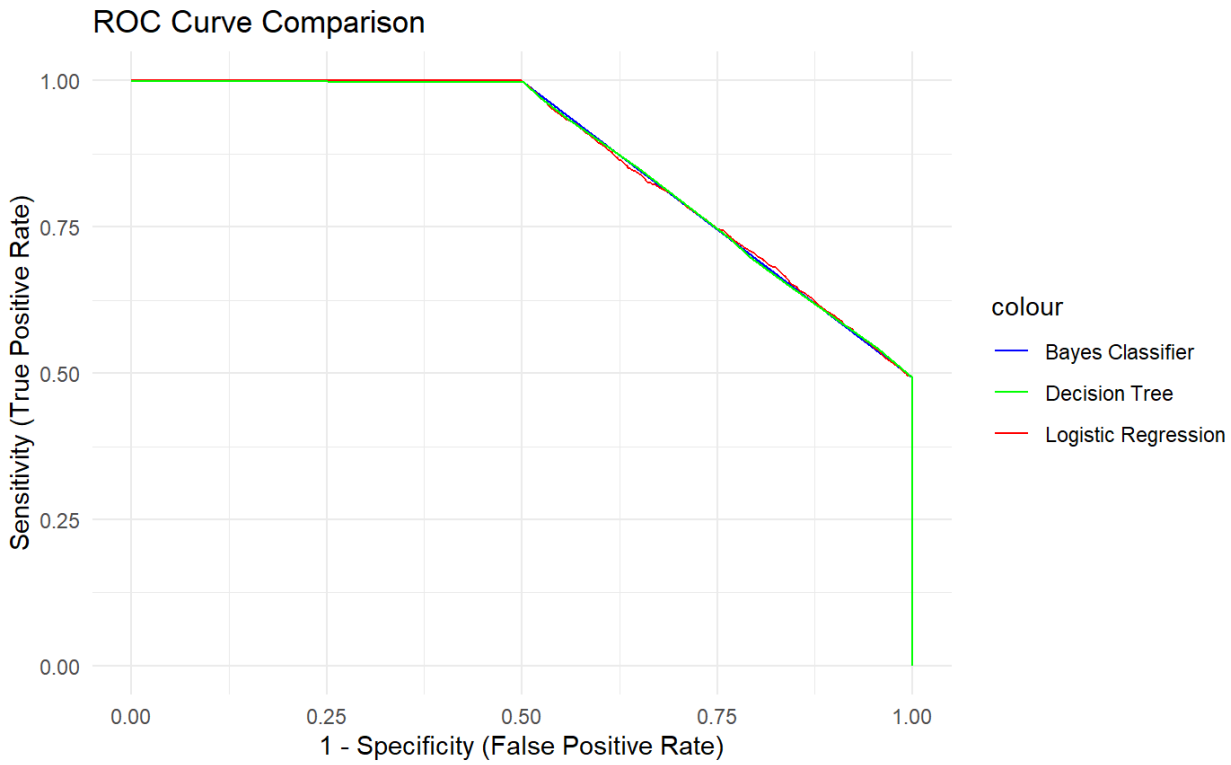
```
Decision Tree Mean Accuracy: 0.74987
```

```r
cat("95% Confidence Interval:", round(ci_tree_lower, 5), "-", round(ci_tree_upper, 5), "\n"
```

```
95% Confidence Interval: 0.74978 - 0.74995
```

Interesting to see how MonteCarlo simulations on Decision Tree Classifier led to the lowest Accuracy score among the 3 classifiers we tested, with the upper bound not even reaching the

```r
library(pROC)

# Compute probabilities for each classifier
data_test$bayes_prob  = r(data_test$x)  # Probabilities from Bayes Rule
data_test$logit_prob = predict(logit_model, newdata = data_test, type = "response")  # Logi
tree_probs = predict(final_tree_model, newdata = data_test, type = "prob")
data_test$tree_prob = tree_probs[,2]  # Take probability of class "1"

roc_bayes = roc(data_test$y, data_test$bayes_prob)
roc_logit = roc(data_test$y, data_test$logit_prob)
roc_tree = roc(data_test$y, data_test$tree_prob)

auc_bayes = auc(roc_bayes)
auc_logit = auc(roc_logit)
auc_tree = auc(roc_tree)

cat("auc_bayes", auc_bayes, "\n")
cat("auc_logit", auc_logit, "\n")
cat("auc_tree", auc_tree, "\n")

# Plot ROC Curves
ggplot() +
  geom_line(aes(x = roc_bayes$specificities, y = roc_bayes$sensitivities, color = "Bayes Cl
  geom_line(aes(x = roc_logit$specificities, y = roc_logit$sensitivities, color = "Logistic
  geom_line(aes(x = roc_tree$specificities, y = roc_tree$sensitivities, color = "Decision T
  labs(
    title = "ROC Curve Comparison",
    x = "1 - Specificity (False Positive Rate)",
    y = "Sensitivity (True Positive Rate)"
  ) +
  scale_color_manual(values = c(
    "Bayes Classifier" = "blue",
    "Logistic Regression" = "red",
    "Decision Tree" = "green"
  )) +
  theme_minimal()
```



Initially, all three models maintain a high sensitivity (~1.0) with low false positives. As the threshold is adjusted, the models begin trading off sensitivity for specificity.

We want to extend the paired t-test to compare all three classifiers. since we have three dependent accuracy scores for each iteration, we can't use the same paired t-test done previously, but there are 2 options: - Paired one-way ANOVA (Repeated Measures ANOVA): tells us if there's a difference

```
library(tidyr)

accuracy_df = data.frame(
  Iteration = rep(1:M, times = 3),
  Model = rep(c("Bayes", "Logistic", "Tree"), each = M),
  Accuracy = c(bayes_accuracies, logit_accuracies, tree_accuracies)
)

# Perform Repeated Measures ANOVA
anova_result = aov(Accuracy ~ Model + Error(Iteration/Model), data = accuracy_df)
summary(anova_result)
```

```
Error: Iteration
          Df    Sum Sq   Mean Sq F value Pr(>F)
Residuals  1 4.564e-05 4.564e-05

Error: Iteration:Model
      Df    Sum Sq   Mean Sq
Model  2 0.0001817 9.084e-05

Error: Within
             Df Sum Sq   Mean Sq F value Pr(>F)
Model         2 0.0000 6.060e-07   0.032  0.968
```

```
Residuals 29994 0.5664 1.888e-05
```

The ANOVA results suggest that there is no statistically significant difference between the three models' accuracy

In case we would have found a difference, the test would have not told us WHERE, so we needed to switch back to Pairwise T-tests with Bonferroni Correction. Luckily it's not the case